Informatique — MPSI/PCSI, 2024–2025

Correction du TP n° 6

Exercice 6.1 (Type d'images, manipulations élémentaires)

1. L'initialisation du code, avec quelques anticipations, s'écrit :

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
# Répertoire de travail
import os
os.chdir("...images")
```

dans le *shell* on obtient

```
>>> os.listdir()
['lena.png', 'RGBA2.png', 'RGB1.png', 'RGBA1.png', 'des.png', 'fleur.png',
'phare.png', 'bateau.png', 'glace.png']
```

2. La définition par compréhension d'une liste de tableaux d'images à partir de la commande donnée dans le sujet s'écrit simplement :

```
images=[np.asarray(Image.open(k+".png")) for k in ["lena","fleur","des"]]
```

On notera juste l'économie du `".png"` concaténé une seule fois.

3. La fonction permettant de définir le type de l'image est la suivante :

```
def TypeImage(image):
    dim = np.shape(image)
    if len(dim) == 2:
        typeimg="N&B"
    elif dim[2]==3:
        typeimg="RGB"
    else:
        typeimg="RGBA"
    return(typeimg, dim[0:2])
```

et utilise la fonction `shape` de la bibliothèque `numpy`. Notez que nous aurions pu simplifier le code sachant

```
len(np.shape(image)) == len(image.shape) == image.ndim
```

4. La procédure permettant d'afficher une image aussi bien en couleur qu'en niveaux de gris peut être définie par :

```
def Affichage(image):
    dim = np.shape(image)
    plt.figure()
    if len(dim)==2:
        plt.imshow(image, cmap="gray")
    else:
        plt.imshow(image)
    plt.axis('off')
    plt.show()
```

5. La procédure permettant d'afficher les composantes d'une image doit identifier le type d'image. Comme les images RGB et RGBA partagent 3 types composantes, il convient de simplifier le code pour en tenir compte. Une façon de le faire peut être :

```
def AffichageComposantes(image):
    plt.figure()
```

```
    if TypeImage(image)[0]=='N&B':
        plt.imshow(image, cmap="gray")
        plt.title('Composante N')
    else:
        if TypeImage(image)[0]=="RGB" or \
        (TypeImage(image)[0][3]=='A' and image[:,:,3].min()==255):
            L,C,A=1,3,0
        else:
            L,C,A=2,2,1
        plt.subplot(L,C,1)
        plt.imshow(image[:,:,0], cmap="Reds")
        plt.title('Composante R')
        plt.subplot(L,C,2)
        plt.imshow(image[:,:,1], cmap="Greens")
        plt.title('Composante G')
        plt.subplot(L,C,3)
        plt.imshow(image[:,:,2], cmap="Blues")
        plt.title('Composante B')
        if A==1:
            plt.subplot(L,C,4)
            plt.imshow(image[:,:,3], cmap="gray")
            plt.title('Composante alpha')
    plt.show()
```

6. Avec une simple coupe et en quelques itérations, on trouve que la commande

```
Affichage(images[0][250:280,230:300])
```

permet de n'afficher que l'œil gauche de l'image `lena.png`.

Exercice 6.2 (Compression d'image)

1. Pour réaliser la compression d'une image en changeant sa résolution, il suffit de parcourir le nombre maximal de motifs indépendants de $N \times N$ pixels contenus dans l'image et d'associer à chacun un nouveau pixel dont la valeur sera la moyenne des valeurs du motif. En veillant à parcourir successivement toutes les cartes de couleurs, il vient le code naïf :

```
def Resolution(image, N):
    dim = np.shape(image)
    newl = dim[0]//N
    newc = dim[1]//N
    if len(dim)==2:
        newimg=np.zeros((newl,newc), dtype='uint8')
        for i in range(newl):
            for j in range(newc):
                value = 0
                for k in range(N):
                    for l in range(N):
                        value += image[N*i+k,N*j+l]
                newimg[i,j] = value//N**2
    else:
        ncolor=dim[2]
        newimg=np.zeros((newl,newc,ncolor), dtype='uint8')
        for i in range(newl):
            for j in range(newc):
                for color in range(ncolor):
                    value = 0
                    for k in range(N):
                        for l in range(N):
                            value += image[N*i+k,N*j+l,color]
                    newimg[i,j,color] = value//N**2
    return(newimg)
```

On commencera par remarquer que, pour chaque carte de couleur (sous-tableau 2D), le code est identique à celui d'une image en niveau de gris et qu'il est donc possible de faire un appel récursif. En utilisant de plus les coupes glissantes de tableaux par pas de N pixels, et la méthode `mean()`, il vient :

```
def Resolution(image, N):
    dim = np.shape(image)
    newdim = np.array(dim)
    newdim[0:2] = np.floor(np.array(dim[0:2])//N)
    newimg=np.zeros(newdim, dtype='uint8')
    if len(dim)==2:
        for i in range(newdim[0]):
            for j in range(newdim[1]):
                newimg[i,j] = int((image[N*i:N*(1+i),N*j:N*(1+j)]).mean())
    else:
        for k in range(dim[2]):
            newimg[:,:,k] = Resolution(image[:,:,k], N)
    return(newimg)
```

où nous avons pris soin de convertir chaque niveau de luminance en entier avec la fonction `int` et où nous avons utilisé la fonction `np.floor` pour prendre la partie entière de chaque élément d'un tableau de type `ndarray`.

2. Pour définir la procédure `AffichageDuo`, on peut exploiter le travail fait pour la procédure `Affichage` en faisant simplement un parcours sur les images du tuple donné en argument.

```
def AffichageDuo(T):
    plt.figure()
    for i,e in enumerate(T):
        plt.subplot(1,2,1+i)
        if e.ndim==2:
            plt.imshow(e, cmap="gray")
        else:
            plt.imshow(e)
        plt.axis('off')
    plt.show()
```

On a utilisé `enumerate` pour réaliser simultanément un parcours par indice (nécessaire au positionnement de l'image avec la commande `subplot`, commençant à 1) et par élément(plus simple comme argument de `imshow`). L'exploitation est directe avec un parcours par éléments.

```
for e in images:
    AffichageDuo((e, Resolution(e, 10)))
```

3. La définition de la procédure `CompressionImage` ne pose pas de difficulté particulière mis à part la détermination de N à partir du taux minimal de compression τ selon :

$$N^2 \geqslant \tau \iff N \geqslant \sqrt{\tau} \iff \lfloor -\sqrt{\tau} \rfloor \geqslant -\sqrt{\tau} \geqslant -N \implies N \geqslant \lceil \sqrt{\tau} \rceil$$

où $\lceil \bullet \rceil = -\lfloor -\bullet \rfloor$ est la fonction partie entière supérieure que l'on appelle avec la fonction `ceil` de la bibliothèque `numpy`, la partie entière inférieure étant définie par la fonction `floor`. Cette procédure s'écrit simplement :

```
def CompressionImage(image, taux, fichier):
    Image.fromarray(Resolution(image, int(np.ceil(np.sqrt(taux))))).save(fichier)
```

4. La fonction de compression fonctionne bien pour tous les types d'images. Le changement de résolution tronque les bords bas et de droite de respectivement

$$n - N \times \left\lfloor \frac{n}{N} \right\rfloor \qquad \text{et} \qquad p - N \times \left\lfloor \frac{p}{N} \right\rfloor$$

pixels avec n le nombre de lignes, p le nombre de colonnes et N^2 le facteur de compression réel. Le nombre total de pixels « oubliés » est donc de :

$$N \times \left(p \times \left\lfloor \frac{n}{N} \right\rfloor + n \times \left\lfloor \frac{p}{N} \right\rfloor - N \times \left\lfloor \frac{n}{N} \right\rfloor \times \left\lfloor \frac{p}{N} \right\rfloor \right)$$

Exercice 6.3 (Filtrage spatial)

1. Les deux fonctions `FMoy` et `FGauss` s'écrivent respectivement :

```
def FMoy(i,j,N):
    return(1/N**2)
def FGauss(i,j,N):
    sigma = (N-1)/6
    return(np.exp(-(i**2+j**2)/sigma**2)/(2*np.pi*sigma**2))
```

2. La fonction `FiltrageConvolution` s'écrit exactement comme dans le sujet et il suffit simplement d'ajouter deux boucles pour parcourir tous les pixels de l'image ; c'est à dire :

```
def FiltrageConvolution(L,N,F):
    k=(N-1)//2
    (n,p)=np.shape(L)
    LF=np.zeros((n,p),dtype='uint8')
    for x in range(0,n):
        for y in range(0,p):
            S=0
            for i in range(-k,k+1):
                for j in range(-k,k+1):
                    if x-i>=0 and x-i<=n-1 and y-j>=0 and y-j<=p-1:
                        S=S+L[x-i,y-j]*F(i,j,N)
            LF[x,y]=S
    return(LF)
```

Ce qui peut aussi s'écrire directement (sans test) :

```
def FiltrageConvolution(L,N,F):
    k=(N-1)//2
    (n,p)=np.shape(L)
    LF=np.zeros((n,p),dtype='uint8')
    for x in range(k,n-k):
        for y in range(k,p-k):
            for i in [w-k for w in range(N)]:
                for j in [w-k for w in range(N)]:
                    LF[x,y]+=L[x-i,y-j]*F(i,j,N)
    return(LF)
```

3. Pour définir le filtre médian, il suffit pour chaque pixel de l'image de calculer la médiane du voisinage de dimension N^2 centré autour du pixel ; ce qui s'écrit :

```
def FiltrageMedian(L,N):
    k=(N-1)//2
    (n,p)=np.shape(L)
    LM=np.zeros((n,p),dtype='uint8')
    for x in range(0,n):
        for y in range(0,p):
            S=[]
            for i in range(x-k,x+k+1):
                for j in range(y-k,y+k+1):
                    if i>=0 and i<=n-1 and j>=0 and j<=p-1:
                        S.append(L[i,j])
            LM[x,y]=np.median(S)
    return(LM)
```

Ce qui peut aussi s'écrire directement (sans test) :

```
def FiltrageMedian(L,N):
    k=(N-1)//2
    n,p=np.shape(L)
    LM=np.zeros((n,p),dtype='uint8')
    for x in range(k,n-k-1):
```

```
        for y in range(k,p-k-1):
            LM[x,y]=np.median((L[x-k:x+k+1,y-k:y+k+1]))
    return(LM)
```

La seule difficulté dans l'implémentation de la fonction de filtrage médian aurait pu être de définir une fonction médiane

```
def mediane(L):
    M=L.copy()# pour ne pas modifier la liste L
    M.sort()# tri par valeurs croissantes
    p=len(L)//2
    if len(L)%2==0:
        return(M[p-1]//2+M[p]//2)
    else:
        return(M[p])
```

qui scinde un intervalle en deux parties identiques, d'où la moyenne (arithmétique modulaire) de deux valeurs dans le cas d'une longueur de liste paire, appliquée pour travailler sur des entiers codés sur un nombre de bits donnés et éviter les *overflow* et, évidemment, renvoyer un entier.

4. L'application des différents filtres pour les tailles $N = 3$ et $N = 5$ aux images `bateau.png` et `phare.png` montre que le filtre médian est *a priori* le plus efficace. Toutefois, plus la taille est grande plus l'image filtrée devient floue. Le choix d'un filtre pour réduire le bruit est donc une affaire de compromis.

Exercice 6.4 (Optimisation du contraste)

1. Deux approches sont possibles pour définir la fonction `BrillanceContraste` :
   — une approche modularisée en définissant les trois fonctions respectivement associées à la brillance B, au contraste C et au contraste de Michelson C_M à partir de leurs définitions et en utilisant les outils de manipulation des listes à partir d'une liste `listeL` comprenant toutes les valeurs du tableau `L`.

```
def Brillance(listeL):
    B=0
    for x in listeL:
        B=B+x
    return(B/len(listeL))
def Contraste(listeL):
    C=0
    B=Brillance(listeL)
    for x in listeL:
        C=C+(x-B)**2
    return(np.sqrt(C/len(listeL)))
def ContrasteMichelson(listeL):
    M=max(listeL)
    m=min(listeL)
    return((M-m)/(M+m))
def BrillanceContraste(L):
    (n,p)=np.shape(L)
    listeL=[L[x,y] for x in range(n) for y in range(p)]
    return(Brillance(listeL),Contraste(listeL),ContrasteMichelson(listeL))
```

La liste `listeL`, de longueur $n \times p$, est ici obtenue par compréhension. Elle aurait aussi pu être obtenue avec deux boucles :

```
listeL=[]
for i in range(n):
    for j in range(p):
        listeL.append(L[i,j])
```

Dans tous les cas cette approche est beaucoup plus longue à programmer que celle qui suit.
   — une approche compacte utilisant les méthodes associées au type `ndarray` : `mean` (qui calcule la moyenne des termes d'un tableau), `min` et `max` (qui renvoient les valeurs minimales et maximales

d'un tableau) et sachant que le carré de toutes les termes d'un tableau de type `ndarray` s'écrit comme pour un flottant avec `**2`. Il vient alors le code :

```
def BrillanceContraste(L):
    B=L.mean()
    C=np.sqrt((L-B)**2).mean()
    CM=(L.max()-L.min())/(L.max()+L.min())
    return(B,C,CM)
```

2. Pour définir la fonction `OptimContraste`, il est nécessaire de définir une fonction permettant de calculer la densité cumulative $\mathcal{P}$:

```
def P(h,k,n,p):
    densite=0
    for i in range(0,k+1):
        densite=densite+h[i]
    return(densite/(n*p))
```

puis de définir la fonction `OptimContraste` qui commence par déterminer le tableau `hist` associé à l'histogramme h avec la ligne de commande donnée dans le sujet puis de définir un tableau `LP` de même dimension que `L` associé à la nouvelle image. On associe ensuite à chaque pixel la nouvelle valeur de luminance renvoyée par la fonction `P` ; soit :

```
def OptimContraste(L):
    hist,bin_edges=np.histogram(L,bins=256,range=[0,255])
    (n,p)=np.shape(L)
    LP=np.zeros((n,p),dtype='uint8')
    for i in range(n):
        for j in range(p):
            LP[i,j]=int(255*P(hist,L[i,j],n,p))
    return(LP)
```

Ce qui peut s'écrire directement en utilisant la méthode `cumsum` du type `ndarray` qui, appliquée à une liste L de longueur n, renvoie la liste de longueur n

$$\left(\sum_{j=0}^{i} L[j]\right)_{i\in[\![0;n-1]\!]}$$

correspondant à la somme cumulée des termes de L. Cette approche permet de calculer la densité cumulative $\mathcal{P}$ sous forme de liste L telle que $L[i] = \lfloor 255 \times \mathcal{P}(i) \rfloor$ avec i dans `[0,len(L)-1]`. Comme précédemment, il convient ensuite de sélectionner une composante pour définir la nouvelle valeur de luminance ; ce qui s'écrit :

```
def OptimContraste(L):
    hist,bin_edges = np.histogram(L,bins=256,range=[0,255])
    (n,p)=np.shape(L)
    LP=np.zeros((n,p))
    P = hist.cumsum()/L.size # « somme cumulée
    for i in range(0,n):
        for j in range(0,p):
            LP[i,j]= int(255*P[L[i,j]])
    return(LP)
```

* *
*